



Application Development Aids



Burroughs

Technical Information Paper

DATE: July 25, 1983

PRODUCT: DMSII (all systems)

Author: Fred Trout
TSC-WEST

SUBJECT: DESIGN PRIMER FOR ON-LINE DATABASE SYSTEMS

This paper will develop general terminology and an overview of the unique characteristics of remote access to a database system. The emphasis will be on data communication requests for database access, functional design, message recovery, and capacity analysis.

Copyright © 1983 Burroughs Corporation

TO: BMG Communications Codes DOC 2 and DOC 4

Printed in U.S. of America

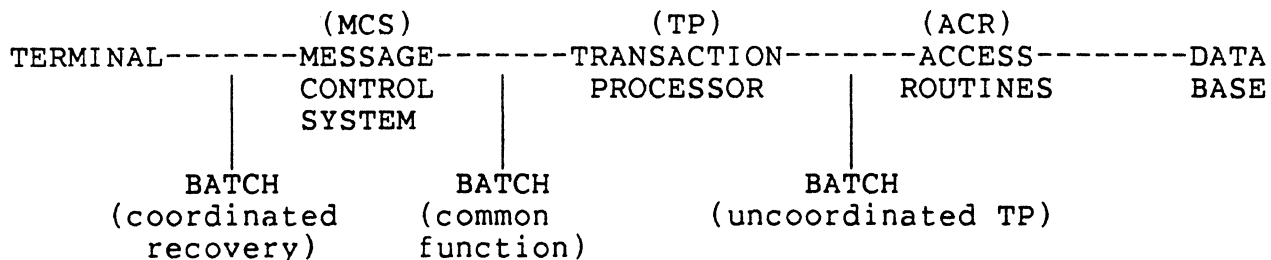
1114238-019

DESIGN PRIMER FOR ON-LINE DATABASE SYSTEMS:

COMPONENTS, INTERFACES, FUNCTIONS, AND OPERATIONS

BASIC COMPONENTS

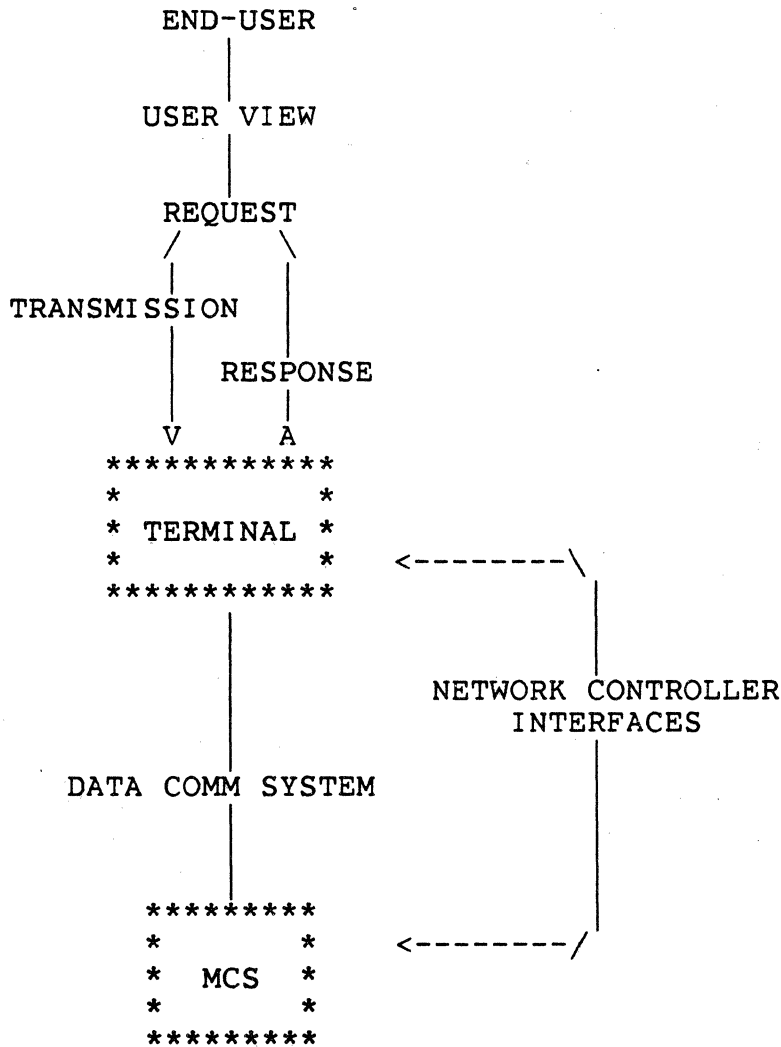
The basic components of an on-line database system are shown in the following diagram. A TERMINAL is the remote device that initiates a request that will require database access. The MESSAGE CONTROL SYSTEM (MCS) receives the request and controls its routing to an application program and other coordinating functions. A TRANSACTION PROCESSOR is application program that has opened the database. The ACCESS ROUTINES provide the access and control of the database as described by the DASDL generation. Note that BATCH users can enter the system at several interfaces, however, the requirement for coordinated recovery usually demands a MCS interface only.



This diagram is the simplex case of a system that is typically many TERMINALS, one or two MCS processes, several TP processes, and one ACR per DATA BASE. However, every component can occur multiple times to form a complex network of processes which provide access to the database(s). The ACCESS ROUTINES for Small Systems are integrated into the MCP; Medium Systems will compile a separate process called a Data Base Program (DBP) for each DATA BASE. These ACRs are interpretive in nature. On Large Systems, the ACR is tailored for each DATA BASE, and its code uses the TP's process.

TERMINAL COMPONENTS

In the following diagram, the TERMINAL node is expanded to include more specific concepts and terminology.



END-USER and USER VIEW

An **END-USER** is a person or process that initiates the access request. The **USER VIEW** is the **END-USER**'s concept of that request. The nature and scope of the **END-USER**'s concept may be different than the actual effect upon the database. For instance, the **END-USER** may consider the request as a single transaction; e.g. add an order. The effect of that request may cause many transactions and database updates to occur, e.g., add the order header record, then add each order line record, and give separate results for each line. The **END-USER** may think of "inventory" as a single file although it is implemented as several related files in the database. **USER VIEW** is how the **END-USER** perceives the structure of the database and the nature and effect of the **TRANSMISSION** and its

corresponding RESPONSE. Properly designed USER VIEWS can give the END-USER a sense of simplicity and stability where the implementation of the procedures, processes, and structures are diverse and flexible.

TERMINAL

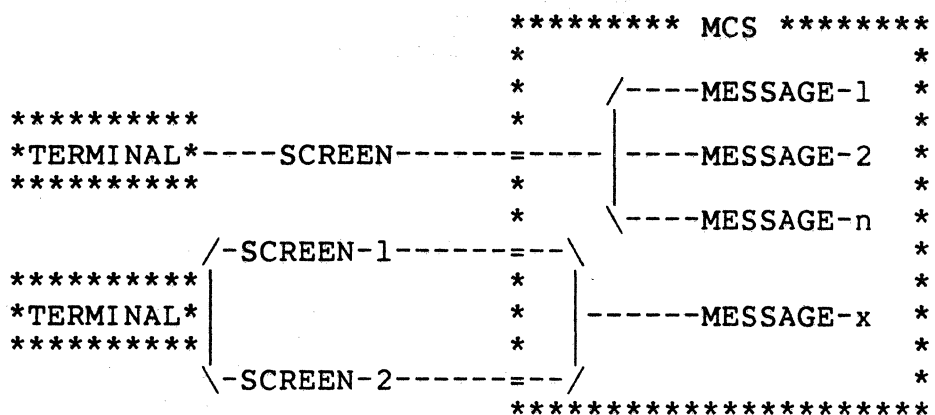
The TERMINAL is the physical device on which the TRANSMISSION is developed and passed to the DATA COMM SYSTEM, and on which the RESPONSE to the request is normally returned. The END-USER and TERMINAL could be a program running in a remote processor or several variations of person, program, processor, and terminal. For the purpose of this discussion, the END-USER will be a reasonably intelligent person using a reasonably unintelligent screen device.

TRANSMISSION AND RESPONSE

The container of communication between the END-USER and the MCS is the TRANSMISSION input to the system or the RESPONSE output from the system. A simple systems design would develop a one-to-one relationship between a TRANSMISSION and its corresponding RESPONSE; however, enhanced USER VIEWS or insufficient terminal capability may require multiple communications on either side of a request interface. In this discussion the term, SCREEN, is used as a synonym for this container.

MESSAGE

The END-USER (with a USER VIEW perception) transmits requests for data base access via TERMINAL (and the datacomm system) to the MCS. Normally, the transmission is a single MESSAGE; however, multiple, perhaps unrelated, MESSAGES could be sent to the MCS in a single transmission. Also, multiple screens may be required to contain a MESSAGE. At this level of interface, a TRANSMISSION is a unit of communication and a MESSAGE is a unit of access request. It would be prudent for the designer of the system, especially in an inexperienced installation, to avoid these more complex communications. However, if a sophisticated USER VIEW is warranted, then the design could include the concept that it is the functional responsibility of the MCS to separate or collect the communication into units of work as in the following diagram.



Output communication is similarly processed by the MCS when completed messages are returned on the RESPONSE interface.

DATA COMM SYSTEM

The remote communications functions between terminals, modems, lines, Data Communications Processors, physical I/O, and Operating System interfaces are typically defined and handled by a NETWORK CONTROLLER system. If the DATA COMM SYSTEM cannot handle these functions, then the MCS or even the TP itself must be responsible for the physical data comm. For this discussion, a NETWORK CONTROLLER system capable of presenting a logical station interface to the MCS is assumed.

MESSAGE CONTROL SYSTEM

A sophisticated MCS, for example, Burroughs GEMCOS, can be organized into general areas of functional responsibility.

```

*****
*   MCS   *
*****
*
* LOGICAL DATA COMM *
*-----*
*
* DIAGNOSTICS        *
*-----*
*
* MESSAGE CONTROL    *
*-----*
*
* MESSAGE AUDIT      *
*-----*
*
* MESSAGE RECOVERY   *
*-----*
*
* TP CONTROL         *
*-----*
*
***** MCS *****

```

The functions of station switching, non-access message switching, screen forming, access security, transmission logging, and other LOGICAL DATA COMM functions will not be addressed in detail. The DIAGNOSTIC functions such as trace, monitor, dump, and statistics will be likewise be ignored. The concentration will be on the MESSAGE accessing the DATA BASE.

Because of the multiple functions of an MCS and the resulting size and complexity, there may exist a requirement for several levels of software interface in the mainframe that perform the MCS functions. Likewise, some of these functions could reside in a datacomm processor described by the network controller interface. This decomposition of the MCS is mentioned only to note its possible existence; this discussion will view the MCS as a single process in the mainframe.

MESSAGE CONTROL, AUDIT, and RECOVERY

The MESSAGE must identify its action, format, and routing. This is typically in the form of a "transaction-id" which implies these characteristics. The basic tool of message level recovery is the MCS AUDIT. The MESSAGE is identified as to its sequence of passing from the MCS and to which TP it was assigned. Then, with the message content,

this information is recorded in a message log or MCS AUDIT. The MCS uses this audit to reprocess the message flow when the database has been recovered to some previous state with a loss of database updates. The MCS does not have to ask the TERMINAL for previous requests to reprocess lost transactions. The MESSAGE and its identification is then passed to (or queued for) a TRANSACTION PROCESSOR.

The user view of the MESSAGE at this level may consist of several operations. Consider the message which requests that a list of employees be changed from status 1 to status 2. If the user views this as a single operation, i.e. if one employee fails the update then none of the employee list should be changed, then the database accessing operations are simplified. The MCS audits the message as single unit and passes it to the TP; the TP processes it in a single transaction state; and the recovery procedures can reprocess the whole message.

If the USER VIEW is designed to consider the same message as multiple, unrelated transactions on each employee, then the message must be decomposed into transactions. Whatever side of the MCS-TP interface does the decomposition, it must consider the recovery consequences. But before discussing this consideration, some further concepts and terminology is required.

TRANSACTION

For this discussion, a TRANSACTION is defined as a set of database changes by a TRANSACTION PROCESSOR. It is required by DMSII that for an audited database these changes occur in TRANSACTION STATE, which is defined by BEGIN TRANSACTION and END TRANSACTION functions in the TP. TRANSACTION STATE is, therefore, one TRANSACTION. When a TP in transaction state fails (ABORT RECOVERY), or the access routines (DBP) or the system fails (HALTLOAD RECOVERY), or disasters happen that require REBUILD or ROLLBACK RECOVERY, some number of TRANSACTIONS may disappear from the "recovered" database. DMSII will recover the data base to some point on the DMSII audit trail where no TP was in transaction state. These QUIETPOINTS may occur naturally, or they may be forced on the audit trail by the SYNCPOINT feature or other high level DMSII functions, e.g., CONTROLPOINT, CLOSE, audit trail change, etc. The lost transactions are those data base changes beyond the QUIETPOINT where the database recovery completed. They have not been recovered because they are in an undetermined state of interaction as in ABORT and HALTLOAD recovery, or they may be transactions beyond a rebuild point, or simply "bad" transactions that have been rolled back.

Also required for updating TPs is a RESTART DATA SET. Its function is to provide a record for each TP; this record will contain information which allows the TP to recreate the current state of the TP that is consistent with the current transaction. Included is the identification of the last completed message. The information in the restart record is provided by the TP. When a database is recovered, the restart record will be maintained to be consistent with the current state of the database. Upon restart or abort the TP reads the restart record, uses the data to recreate its state, and requests from the MCS all subsequent messages. This reprocessing will bring the data base forward to real time when new requests can be processed.

COMPLEX MESSAGE

A simple MESSAGE is one that causes a single TRANSACTION to occur in the TP, whereas, a COMPLEX MESSAGE causes more than one TRANSACTION STATE as in the examples above. A transaction for a simple message would include:

```

For each-message:
    lock and/or create all database resources to be changed,
    update all database record areas,
    update the restart record area, including the
        identification of the message from the MCS,
    if synchronous-recovery then communicate message sequence,
    BEGIN TRANSACTION
        update database via STORE, DELETE, etc., functions,
    END TRANSACTION
  
```

There are two ways to handle a complex message in the MCS. One is to simply decompose it into simple messages to be sent across the interface. The MCS would need to collect the message COMPLETES for the TERMINAL response. Sending a complex message to the TP requires that a restarted TP be able to determine not only which message to begin reprocessing, but which transaction within the message is consistent with the recovered database. This requirement may determine the format of the complex message and increase the complexity of the restart information and restart procedures.

Three examples of the interface from the SCREEN in the MCS to the TRANSACTION in the TP and back is shown in the following diagram.

```

***** MCS *****
*
-->SCREEN:
*   simple MESSAGE---AUDIT----->* TRANSACTION *
<--RESPONSE-----*<--COMPLETE-----< *
*
*****
*
-->SCREEN:
* COMPLEX MESSAGE:
*   simple-----AUDIT----->* TRANSACTION *
*                               *<--COMPLETE-----< *
*   simple-----AUDIT----->* TRANSACTION *
*                               *<--COMPLETE-----< *
*
*   .
*   .
*   .
<--RESPONSE
*****
*
-->SCREEN:
* COMPLEX MESSAGE---AUDIT----->* TRANSACTION *
*                               * TRANSACTION *
*                               *   .   *
*                               *   .   *
*                               *   .   *
<--RESPONSE-----*<--COMPLETE-----< *
*
***** MCS *****
***** TP *****
*
* Case 1
*
*
*
* Case 2
*
*
*
*
*
*
*
*
* Case 3
*
*
*
*
*
*
*
*
*

```

In case 2 the MCS will queue and audit all the simple messages at once, and also be able to collect and save (thru a haltload) the results for the single response. The TP has this responsibility in case 3. The TP will typically have the better tools available to save the results of multiple transactions.

RECOVERY

Note that the unit of MCS AUDIT is the outbound message interface to the TP. This becomes the level and sequence of the reprocessing phase of recovery. Typically, only messages which will require changes to the database are audited. However, if the MCS cannot recognize the updating messages or there are some applicational requirements to audit inquiries, then the reprocessing procedures must handle (typically ignore) the unnecessary inquiries.

The design of reprocessing procedures would be for each TP to request all messages from a specific message forward in sequence as a functional capability of the MCS. In this reprocessing interface, knowledge that these messages have, in fact, been previously completed successfully must be known to the TP. The effect of a reprocessed message must be controlled if any results of the message are outside the database. A second response is likely not necessary or even wanted; in fact, the END-USER or his TERMINAL may not be present in this real-time. The last message sent to a TP that caused an abort is also useful information to pass on this interface in order to avoid a abort loop.

Interfaces which cause multiple results create another recovery complication for the system. Whether to collect results for a single communication or develop a single-thread interface will determine the recovery responsibilities for the outbound communication. Again, prudent designers develop simple or complex interfaces according to their capabilities instead of the requirements. They also must consider the design as a whole, from USER-VIEW input to USER-VIEW output. Complex user-views and SCREEN/RESPONSE interfaces often cause unnecessary complexity and resource consumption in the processes providing those features.

Failures during reprocessing may have a significantly different error handling process, especially if the results of the original transaction have "escaped" the system, e.g., money was released, or inventory was shipped. This loss of the ability to recreate the database negates the guarantee that the database reflects the real-world in real-time. This is critical in some applications, and special operations or reversing transactions may be required.

Transactions of this nature, i.e., "escaped results", are typically implemented so that the TRANSACTION is ended with a database SYNC. This technique provides that database recovery will assure a consistent state with the real-world for abort and haltload recoveries. However, the recoveries which do not recover all the way to the current SYNCPOINT will still require special handling for reprocessing and recovery failures.

Transaction results which affect systems or files outside the database will not be recovered. Output files, printer files, etc., cannot be "backed up" with the database, and may be lost over a haltload. Different application designs must be used. Two-phased process or two processes which update then inquire and report is one possible solution. "Temporary" or "Blackboard" datasets are often used contain output information for these applications.

To maintain some semblance of order during restart recovery, the MCS should first reconstruct the full TP environment and insure that all TPs have requested the message sequence for reprocessing. Then all previously completed messages are reprocessed (single threaded for SYNCHRONOUS RECOVERY). This procedure will bring the data base up to real-time and allow previously queued messages and new requests to be processed.

SYNCHRONOUS RECOVERY

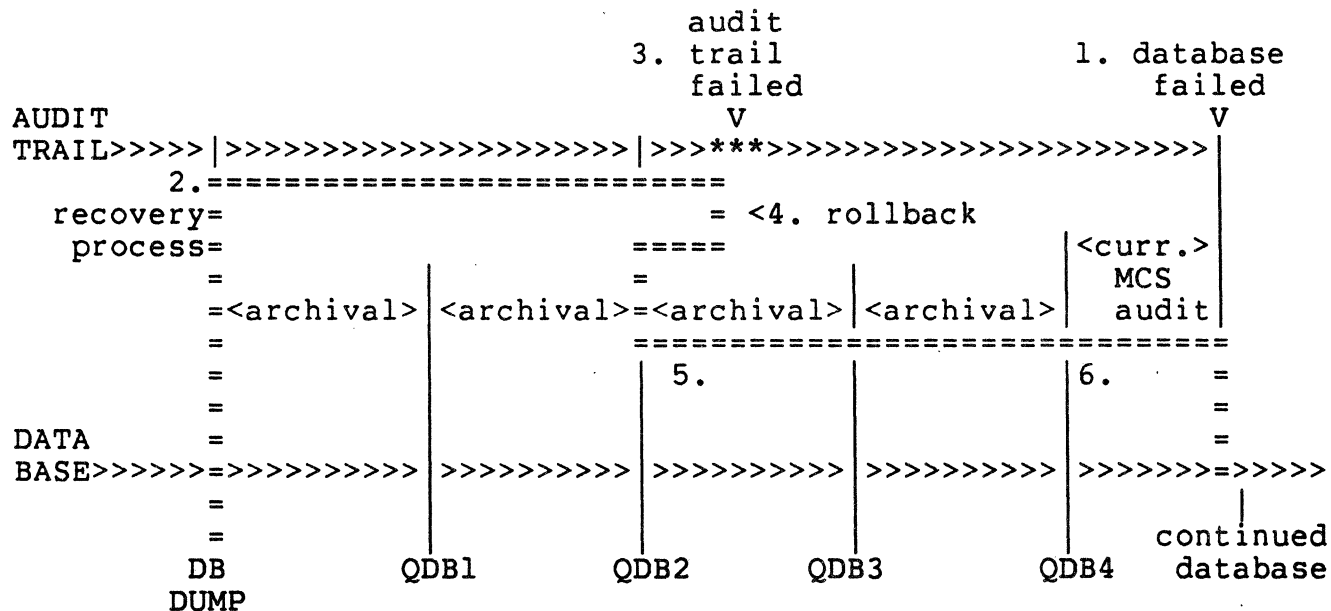
Thus far, there has been no requirement to reprocess the transactions in a sequence which will guarantee the same effect to the database as the original transaction sequence. The sequence has only required that they are presented to a TP in the same order. In order to insure that the database is updated in the same sequence during reprocessing, two-phased transaction processing must be employed. First, the TP must insure that all resources used in the original transaction are locked from other users before entering transaction state. Second, between this logical point and the end of transaction state (resource free), a communication to the MCS must be completed. It is the sequence of this communication that will determine the sequence of reprocessing. The MCS, of course,

must be able to provide this capability called SYNCHRONOUS RECOVERY. It is also a requirement that the reprocessing of messages be single-threaded for all TPs to insure consistent database results. Batch TPs (not invoked by the MCS) which update are not allowed in this environment. Coordinated recovery of all updaters is required for synchronous recovery. The MCS could provide a TRANSMISSION/RESPONSE and a MESSAGE/COMPLETE interface for a "batch" TP in order to provide the message level audit and MCS TP environment needed for coordinated recovery.

ARCHIVAL RECOVERY

There is another type of message recovery which the MCS could provide. For the disaster where the database was unable to recover to the most recent QUIETPOINT, e.g., a bad DATABASE AUDIT TRAIL, or a failure of the rebuild recovery process itself, an ARCHIVAL RECOVERY may save the day (or days). A capability of the MCS could be provided that, from a quiescent (closed) database, a sequence of messages could be reprocessed to form a more recent version. This is usually implemented by a dumping, unloading, or otherwise marking the message audit where it is coordinated with a quiescent database. Although this type of recovery may not, by itself, be capable of reconstructing the database to the current point in time; it might change a major disaster to a minor inconvenience. Archival Recovery can also rebuild a database where a DMSII REBUILD RECOVERY cannot, e.g., a REORGANIZED database.

The following diagram represents a recovery from a failed audit trail. When the database failed (1), a rebuild was attempted from a previous database dump (2) until the database audit trail failed (3). The database was rolled back (4) or rebuilt to the point of QDB2, and the archived messages were reprocessed (5) to the current MCS audit trail. Normal message level recovery (6) reprocessed and continued the database access.



TP CONTROL

--- -----

It is likely that the MCS will also perform some functions of TP (process) CONTROL. As the MCS receives TRANSMISSIONS from the TERMINAL, the TP, for which the message is to be processed, is invoked by the MCS if that process is not already running. The MCS should maintain knowledge of each TP's status as a process, and typically its message queue status. This implies a communication/result interface which was specifically developed above as a MESSAGE and a COMPLETE. A TP's termination whether normal or abnormal must also be known to the MCS.

Each UPDATER TP will create a RESTART record which identifies a "live" TP. These TP's will have a RESTART state (contained in the restart record) which must remain current with the database state for the life of the process. A deletion of the RESTART record would remove the TP from the "TP environment". Therefore, at any previous point in time, the residual information in the RESTART DATA SET could be used by the MCS to re-invoke its TP environment. The message reprocessing could then proceed normally.

Other types of TP control could be implemented by the MCS. BOJ/EOJ, START/STOP, memory and processor PRIORITY changes, or multiple invocations could be developed depending upon the type or volume of messages and the need for unused TP resources.

The MESSAGE routing interface and the corresponding number and capability of TPs can be designed for a variety of criteria. The design of the TP's message processing capabilities may be based upon application modules, transaction type, message load balancing, database target, resource consumption, or other logical systems of message distribution. The DMSII ACCESS ROUTINES are designed, even optimized, to provide a multiple TP environment with no restrictions as to the type or sequence of access all the way down to the record level. System designs which take advantage of this DMSII capability to the limit of their resources will maximize the throughput.

A common error in systems design is not providing enough TP resources to maintain a given response time requirement. A long message process, even with a short transaction state, can bottleneck the system by extended use of a vital code (process) resource. Well designed message routing and queue depth control can resolve the (hopefully) rare case of a long transaction and smooth the effect of request activity peaks.

TRANSACTION PROCESSOR

A TRANSACTION PROCESSOR (TP) is an application program which has opened the database. In an on-line, system the TP's general functional responsibilities are to communicate with the MCS, receive and process MESSAGES, access the database, and send the results back to the MCS. Thus far, most of the discussion has assumed that the message processing required database changes, and therefore, the audit and recovery interfaces were discussed first. However, many of the user's requests in a real-world system are database inquiry requests.

One abstraction of the TP environment categorizes the TPs according to the possibility of a TRANSACTION STATE happening during the processing of a message. First, there are TPs which handle only inquiry messages. They open the database INQUIRY, and having no RESTART state or recovery capability, other techniques of reconstructing the TP environment and message queues must be employed. Second, there are TPs which handle only messages which require data base changes. They open the data base UPDATE, have a RESTART state, and normally guarantee a TRANSACTION STATE on every message. Third, there are TPs which handle both inquiry and update messages. They have the data base opened UPDATE; they have a RESTART state and therefore are involved with data base recovery; but they do not "guarantee" a TRANSACTION STATE for the inquiry messages.

INQUIRY MESSAGES

The handling of inquiry messages must be considered in the design as they affect every component and interface. The major decisions concern what is the END-USER requirement for response to a inquiry during a database recovery and whether TPs process both inquiry and update messages. The basic requirement upon the MCS is that it know the messages (or TPs) which require (or provide) database updates.

The USER VIEW of database inquiry requests are typically implemented with no recovery capability. That is, the system does not put the message into the MCS AUDIT and no TRANSACTION STATE is executed in the TP. When the system recognizes a inquiry has been lost or failed, then a simple "say again" is communicated to the requester.

For the case of a TP which handles inquiry messages only and has opened the database INQUIRY, DMSII will not inform the TP of an abort recovery. The only possible effect of the recovery would be on the rare re-read of a record whose data or existence was backed out. Failures of inquiry TPs can only be recognized by the MCS implementing status checks or communication timeout capabilities. Since this style of TP does not create a RESTART record, other means of reconstructing the INQUIRY TP environment after a haltload recovery must be developed.

Recognizing a failure is obviously clear for recoveries other than abort recovery. All the components know that there has been trouble, but there is residual information, interfaces, and procedures to control and reconstruct the components to a real-time environment. Recognizing an abort failure is a bit more ambiguous.

An abort is caused by the failure (terminate) of a program in TRANSACTION STATE. The ACR, upon receiving this knowledge, has the responsibility of informing the other updating TPs. Note that they are all TPs which have the database opened UPDATE not just the ones which are currently updating. The ACR will prevent a TP from entering TRANSACTION STATE by holding that function request until all other TPs are out of transaction state. The ACR allows the other TPs to complete any current transaction states, and all other requests are held up. The ACR then rolls the database back and marks all TPs "to be aborted". The TPs are allowed to resume, and on a TP's next (or current) BEGIN or END TRANSACTION or CLOSE request the ACR will return the "database aborted" results.

It is important to recognize that each TP is informed individually and only on a BEGIN, END, or CLOSE. An updating TP, handling only inquiry messages (no TRANSACTION STATES) for a period of time, will not know of the abort. It will take a transaction state by each updater to get the whole environment back into real-time. The MCS must implement procedures and/or interfaces to accomplish this since it is probably unacceptable to have an unresolved abort for a long period of time.

The progression of an abort can be described by the abort states of the several components. A TP fails in TRANSACTION STATE and its termination causes an abort state in the ACR. The ACR must hold up TP requests until all TPs are out of transaction state then put the DATA BASE in an aborted state (rolled back to the last QUIETPOINT). The "abort state" is then split across all UPDATE TPs as "to be aborted" and, the TPs are allowed to continue. The MCS does not know of the abort yet; even if it knows about the failing TP via status or timeout, it does not know if that TP caused an abort. A TP could now receive the abort results and inform the MCS of the abort by requesting its previous message(s), but that is not guaranteed to happen, as described above. In fact, the remaining TP(s) may be quiescent, no message traffic at all.

A possible solution to this situation is the development of an "executive" TP. The TP would periodically execute a TRANSACTION STATE simply to get an early warning of the abort. One might find other useful functions for this TP, e.g., the restart dataset interface during recovery or the MCS's status clock. Of course, the MCS could open the database itself, but that may not be feasible in most implementations.

Another, perhaps more feasible, solution is that all updating TPs "guarantee" a TRANSACTION STATE, even on a inquiry message. This technique not only provides for knowledge of an abort on each message/complete interface, but a more recent message-id can be captured in the restart record in the case of a long series of only inquiries.

In the general case, once the MCS recognizes the abort state, the typical implementation provides an immediate message to each (updating) TP that an abort occurred. The TP then must insure that the "to be aborted" state is cleared by attempting a dummy transaction. Message and Complete queues, if any, must be cleared also. Whether the queues are cleared by the MCS or the TP and what happens to those communications is dependant upon several other design variables and a bit complex for this discussion. Since multiple aborts during this abort recovery procedure could cause a more complex interface, the MCS usually holds any reprocessing until all TPs have stabilized, i.e., read the restart record, recreated their memory state, and requested their recovery message.

DATA BASE FUNCTIONS

The interface between the TP and the ACR is the DATA BASE FUNCTION requests. These functions (using COBOL) are the FIND, LOCK, SET, and CREATE verbs used to prepare for the TRANSACTION STATE, and the BEGIN TRANSACTION, STORE, DELETE, INSERT, REMOVE, and END TRANSACTION used to implement a transaction. OPEN, CLOSE, and FREE complete the general function interface list. Each FUNCTION request causes that function to be performed in the ACR and/or DATA BASE and any errors are returned to

the TP via DMSTATUS. Errors or unsatisfied requests change the TP execution sequence via ON EXCEPTION procedures or (for Medium and Large Systems) USE ON DMERROR DECLARATIVE.

A MESSAGE may be a simple inquiry requiring only a FIND function; or it may be a simple update using LOCK, BEGIN, STORE, END functions. It also may be a complex transaction requiring many DATA BASE FUNCTIONS to prepare and execute the transaction. This difference, the number and type of function, becomes important when designing a TP's message handling capabilities and for analyzing the rates and capacities of the system. Simple data base and message designs usually result in short, simple, and predictable DATA BASE FUNCTION requests.

ACCESS ROUTINES

Another "explosion" of work may be created by the ACCESS ROUTINE's handling of a single FUNCTION. The unit of this work is a DATABASE OPERATION. A FIND NEXT on a DATASET produces a single database operation, a read; similarly, a BEGIN TRANSACTION normally just sets a state within the ACR. A random FIND via set could be simple or complex depending upon the type of set, the complexity of the key and the set selection expression, the population of the set, and several other variables. The DELETE or add of a dataset record must also insure the all references to that record are removed from or inserted into every spanning set or automatic subset. When many sets are defined for a dataset, then many implicit removes or inserts will be required. An END TRANSACTION implicitly frees all the locked records for the TP and typically stores the restart record. These later FUNCTIONS create multiple DATABASE OPERATIONS.

That is probably enough examples to show that there is a wide range of processing done by the ACR for a single function. The number of database operations is partly dependant on the type of function and the implicit functions it may produce. But it is mainly due to the user-specified variables in the DASDL design, the user-specified criteria in the function, the user requirements for database size, and the user's physical resources available to the ACR. Simple data base structure design and simple function requests by that user will usually result in predictably few database operations.

LOGICAL AND PHYSICAL I/O

The variables above are mainly concerned with the amount of processing that is required. There is yet another variable, often having a greater affect, and that is the amount of waiting that is required. The subject of this waiting time is physical I/O. For each access to the database (logical I/O), whether for pointers, controls, or user-data, a physical I/O may be required. Although DMSII ACRs can multi-thread TPs and maximum overlap capability was a major design criteria, there is a limit to the effectiveness of I/O overlap. The criteria controlling the requirement for a physical I/O are, again, site-oriented and user-specified. Beyond the ACR's request for a physical I/O, is an additional wait-variable of I/O path and spindle contention. At the

lowest level, there is the bit-fiddler's standbys, data transfer rates, seek time, and rotational delay which should remain beyond the DBA's manipulation, but their effect should be considered.

TOTAL SYSTEM COMPONENTS AND INTERFACES

The following diagram develops the components and interfaces for an on-line DMSII database accessing system.

COMPONENT -----	INTERFACES -----		COMPONENT -----
	DIRECT (DATA)	INDIRECT	
=====	=====	=====	=====
END-USER	USER VIEW		TERMINAL
-----	-----	-----	-----
TERMINAL	>> TRANSMISSION RESPONSE <<	* DATA COMM SYSTEM	MCS
-----	-----	-----	-----
MCS	>> MESSAGE COMPLETE <<	* TP CONTROL * ROUTING * FAILURES * RECOVERY	TRANSACTION PROCESSOR
-----	-----	-----	-----
TRANSACTION PROCESSOR	>> DB FUNCTION >> RECORD << DMSTATUS <<	* TRANSACTION STATE	ACCESS ROUTINES
-----	-----	-----	-----
ACCESS ROUTINES	> LOGICAL I/O <	* AUDIT * DB OPERATIONS * RECOVERY	DATA BASE
-----	-----	-----	-----
DATA BASE	>PHYSICAL I/O<		DISK
-----	-----	-----	-----

CAPACITY ANALYSIS

At system design time there is always a concern for the overall effectiveness of response to the end-user. Typically, these measures are coached in terms of "transaction rates" or "response times". But what is a "transaction"? Or at what interface was the "response" measured? All too often these rates and response times are not, themselves, properly defined. Even then, their values or requirements are unrealistic for given resources. Early design decisions cry out for some real numbers, but how does one produce realistic numbers from the multitude of interactive variables that are found in such a complex system?

It is certainly obvious that at each site, for each design, with each optimization tradeoff, different response characteristics will be produced. Also, the measurement cannot be accurate until many of these variables have been implemented. Assuming there is some minimum threshold of response requirement, there are two major variables that can be the basis of overall strategy; one is the system's requirements, and the other is the systems resources.

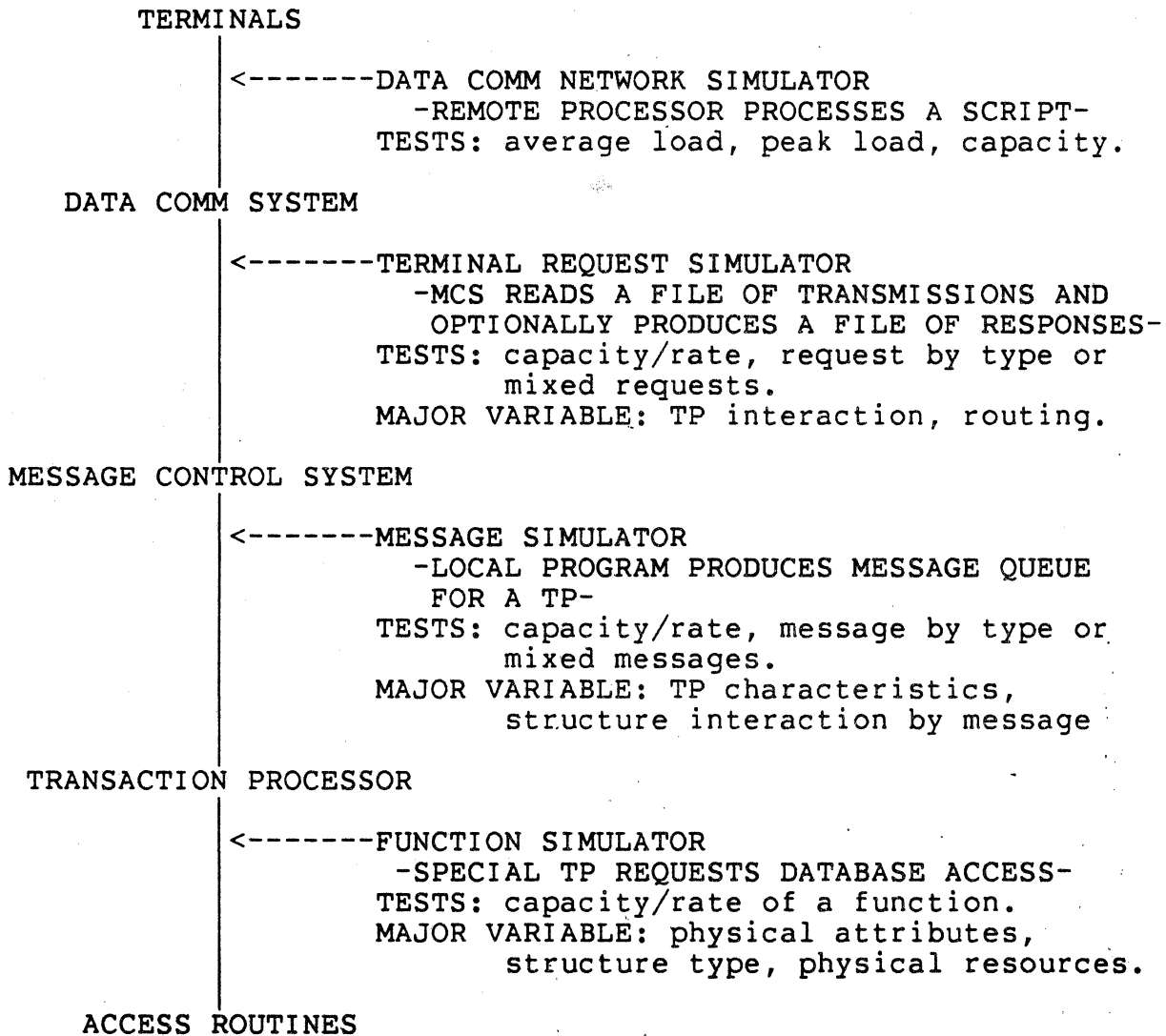
If one holds the system's requirements a constant, then the strategy must include a design that will be flexible in terms of resources to provide varying levels of response. DMSII is designed to take advantage of additional resources (buffers, memory, I/O paths, etc.) according to several specifications by the designer, but that ability is limited by the overall design. DMSII is also designed to a single standard across its hardware systems, but the difference in implementations and architectures affect many of the variables described above. Simple systems designed with simple structures and standard interfaces will be easiest to convert to a future expansion of hardware. The standard interfaces should include not only the Burroughs Products, GEMCOS, NDL-II, COBOL-74, etc., but standard user accessing routines, error routines, recovery procedures, and compatible interprocess communication capabilities.

If the hardware resources are the constraint, then a strategy of developing simple, expandable requirements would be used to find an optimum capacity. The idea here is one of starting small and developing the on-line system's capability according to the capacity of the resources. The real danger in overdesign when hardware resources are limited is that the end-user's expectations of an application are severely disappointed. The end-user's frustration with unreasonable response time or limited function can have a disastrous effect on future planning, control, and cooperation. Hell's furies are also unmatched by disappointed users.

Usually there is some middle ground or an envelope of requirements and resources with some of the sides more flexible than others. For example, disk space is probably more expandable than memory or processor; response time more flexible than functional requests. In any case, system models should be developed, analyzed, and optimized reiteratively. These models should be implemented first with minimum requirements and expanded according to the learning curve of the

system's capacity.

This test environment provides the characteristics of the total system. If capacity simulators are developed at each interface, the characteristics of each component also be analyzed. One might visualize such a modeling environment as:



The reason for developing a modeling or testing system is the requirement for information. Information is needed for design and resource decisions. Accurate information should produce better decisions and therefore more effective systems. This information is first used as the building blocks or learning curve in order to help in the decisions necessary for initial system designs. Each measurement becomes the basis of comparison for measuring the effect of the next change.

The applications, database, network, and hardware are never static systems. A well developed modeling system can be an valuable resource to provide accurate, actual information as the future systems are developed or explored. It can find bottlenecks to be optimized; it can find available resources to be utilized; it can explore possible alternatives; and it can measure the effect of design decisions.

In order to be able to take advantage of the information a modeling system provides, the DBA must plan its development similar to any other project. That is, he must provide for design and programming resources, production (testing) machine resources, and learning curves of the DMSII statistics and operating systems logs for analyzation.

The on-line systems development should begin with simple structure, interface, and component designs. The characteristics of the system are then tested for bottlenecks and optimized. The capacities and available resources are identified for future development. Whether to reiterate the process to find the most effective system is questionable in a volatile application or hardware environment. Perhaps only a "satisfactory" system is needed as the next change will introduce a large set of variables to be analyzed. The important product is knowledge; through knowledge, design; through design, effective on-line systems.

Table of Contents

DESIGN PRIMER FOR ON-LINE DATABASE SYSTEMS:	1
BASIC COMPONENTS	1
TERMINAL COMPONENTS.	2
END-USER and USER VIEW	2
TERMINAL	3
TRANSMISSION AND RESPONSE.	3
MESSAGE.	3
DATA COMM SYSTEM	4
MESSAGE CONTROL SYSTEM	5
MESSAGE CONTROL, AUDIT, and RECOVERY	5
TRANSACTION	6
COMPLEX MESSAGE	7
RECOVERY.	8
SYNCHRONOUS RECOVERY.	9
ARCHIVAL RECOVERY	10
TP CONTROL	11
TRANSACTION PROCESSOR.	11
INQUIRY MESSAGES	12
DATA BASE FUNCTIONS.	13
ACCESS ROUTINES.	14
LOGICAL AND PHYSICAL I/O	14
TOTAL SYSTEM COMPONENTS AND INTERFACES	16
CAPACITY ANALYSIS.	17